Rational's Experience Using Ada[1] for Very Large Systems

James E. Archer, Jr.
Michael T. Devlin

1. Introduction

The development of very large software systems challenges human intellect and creativity. It has been described as one of the most complex activities undertaken by Man. The risks associated with such an effort are increased by its size and by the involvement of participants from different organizations, locations, and, possibly, countries. By any measure, the software for Space Station ranks among the most ambitious projects ever undertaken.

Considerable research effort has been devoted to solving the problems involved in the construction of such large systems. Unfortunately, while much of the resulting technology is available in the literature, it is not widely used [16]. Reducing theory to practice is always difficult; the rate at which this has been accomplished for software seems particularly discouraging. These difficulties prompted the Department of Defense to start the STARS program [7] and to establish the Software Engineering Institute [3] to focus on improving the state of practice.

2. Motivation

In 1981, Rational[2] set out to produce an interactive environment that would improve productivity for the development of large software systems. The mission was to create an environment that supported high-level language development of systems developed using modern software engineering principles. This mission was built on the belief in recent advances in programming languages, methods, and environments.

In designing the Rational Environment[2], object-oriented design [4], abstraction [9], information hiding [5], and reusability [10] were important both in terms of use in our design and as methods to be supported. Prototyping [11] was of particular importance because it gave access to the advantages of the environment and its component technologies, at the earliest possible time.

The language to be supported was Ada. This was an easy choice. Ada appeared to be the latest and best engineered language for building large systems [1]. In particular, the separation of specification from bodies appeared to offer a real advantage: it allowed the language to be used during design, as well as implementation, and it supported a realistic opportunity for reusability [8].

Experience with research programming environments had shown that access to a set of integrated facilities could greatly leverage the ability of individuals to produce systems. The most widely used of these environments supported interpretive or dynamically typed languages, most notably Lisp [13]. Research efforts to support more appropriate, strongly typed languages were interesting, but they centered mostly on interpretive implementations for student subsets [2, 14]. Even so, the benefits of these systems

---

[1] Ada is a registered trademark of the U.S. Government Ada Joint Program Office.

[2] Rational and Rational Environment are trademarks of Rational.

suggested the feasibility of building a compilation-based environment for team development of large systems.

From the outset, it was clear that the Environment itself was an example of the kind of system whose development it was intended to facilitate. Although it would not be possible to use the Environment early in its construction, the other central technology themes, language and methods, were still available. To support these before the environment was functional, a set of tools was constructed to support Ada development in a conventional, batch-oriented manner. We don't think of the resulting tool set as an "environment"; however, it does constitute an APSE in the Stoneman sense [12], it includes a validated compiler, and it is comparable to other commercially available Ada compilation systems. The development of this tool set involved more than 300,000 lines of Ada code; building it helped to improve our understanding of the problems and opportunities associated with the evolution of the Rational Environment.

3. Environment Characteristics

The Rational Environment is the operating software for the Rational R1000, a time-shared computer implementing a proprietary, Ada-oriented architecture. It is written entirely in Ada, with considerably less than 1% of its statements being machine code insertions.

The Environment is the system interface; all users of the system use the same facilities. Although general-purpose computing is well supported, the system is designed to be used by project-related personnel with some interest in and facility with Ada and programming language concepts.

3.1. Ada Framework

The Environment directory structure is hierarchically organized. Names in this structure are Ada simple identifiers separated by periods, as with Ada qualified names. This structure contains a variety of objects of a variety of system and user-defined types. One common object type is *file*; another is *Ada*. Files resemble files on any other system. Ada objects are more interesting.

An Ada unit under development is an Ada object. The name of the object is the name of the unit that it represents. Ada objects corresponding to library units have two parts: visible part and body. Separate subunits of Ada units are children of their parent Ada unit and are named as such. As a result, the same name is used to refer to the unit when it is edited, compiled, and executed. All of the units residing in the same directory substructure constitute an Ada library, and there are provisions for creating libraries, hierarchical or otherwise, from multiple simple libraries.

The treatment of Ada units as typed objects is central to the design of the Rational Environment. In addition to supporting an object-oriented view of the unit throughout the compilation process, the storage of the program object as an attributed DIANA tree [6] provides access to the program structure in a way that makes a variety of interesting facilities available. These include Ada-specific editing operations, incremental compilation, compilation ordering and interconnection facilities, and direct execution of Ada statements.

3.2. Compilation

The traditional compilation model involves reading files of program source into a series of tools that produce various processed forms of the original program. During this process, new objects with new names are created and the user is forced to track the correspondence between the current program text and the current executable version. In contrast, the Environment compilation model centers on Ada units as definable objects that are transformed by editing and compilation between three principal states: source, installed, and coded.[3]

A source unit has been parsed, but has yet to be compiled. It isn't just another form of file: it's a DIANA tree sufficient to support interactive syntax checking and to perform operations that depend on the structure of the unit. Maintaining this structure makes it convenient to keep units syntactically consistent at all times, greatly reducing the time lost trying to compile units with syntax errors. Ada was explicitly designed to have a declarative structure that facilitates the expression of complex system interaction in a way that can be statically checked. Installed units have passed the semantic checks necessary to assure that they are consistent, both internally and with units that they reference. Getting units semantically consistent and keeping them consistent is one of the major programming activities in Ada development. Once a unit is installed, coding is just a matter of time and computation required to get into execution; there is no intellectual effort involved. Coded units are ready for execution. Programs are intended to be executed, so this is the final state in the compilation process, if not the most interesting one. The existence of separate installed and coded states reduces the compilation effort, increasing interactivity during one of the challenging parts of the programming process.

The Environment supports a spectrum of compilation paradigms:

- Batch installation and coding with fully automatic ordering
- Editor-based installation and coding of individual units
- Incremental, statement/declaration-level changes to installed units

All these paradigms make use of the system's knowledge of the structure of the units being processed to determine correctness and compilation ordering. Incremental changes to compiled units has an immediate intuitive appeal regardless of the language involved. Making small changes and only recompiling what has actually changed reduces both the total compilation effort and the time between a change and getting the benefit from that change. This is particularly important for Ada: getting immediate feedback on the legality of a change makes it possible for the developer to use the declarative structure more effectively in evolving the program. Early detection of problems minimizes wasted effort.

Another benefit to be derived from incremental operations is the ability to add new functionality to a specification with minimal compilation effort. The goal is to add declarations to the visible part of a package without allowing illegal changes or requiring clients of the package to be recompiled -- all of the benefits of strong typing without the consequences. Providing this facility was one of the more interesting technical challenges in building the Environment [15], but it was certainly worth the effort.

Immediate semantic information about programs under development is not limited to the compilation process. Part of providing incremental semantics was building a database of

---

[3] Compilation for targets other than the R1000 may involve more than these three states.

declaration-level dependency information sufficient, in conjunction with the DIANA trees, to determine the legality and impact of incremental changes. This information turns out to be generally useful. For installed Ada units, the relationships between declarations and their uses is a matter of great interest. Given the rich structure of Ada naming (use clauses, renaming declarations, overloading), it isn't possible, much less desirable, to keep track of these relationships on the basis of the program text. Using the compilation dependency information, these relationships can be checked interactively.

Definition is the name of an operation to show the declarations of an object that is referenced somewhere in an Ada unit. As typically used, the user points to the reference of interest[4] and presses the key to provide its definition. The declaration of the object referenced is brought onto the screen in the context of its Ada unit. It is also possible to find the implementation of the declaration. Definition is very useful in refreshing familiar code in the user's mind; it is invaluable in understanding unfamiliar code. A generalization of this mechanism to all system objects is the basic command for visiting objects of any type, traversing the directory structure, and changing context.

Show_Usage is the name of the operation that goes in the opposite direction: it provides the set of references shared by a declaration, a form of interactive cross-reference. If only one unit references the declaration, the referencing unit is brought onto the screen with each of the references underlined in a way that it is easy to traverse from one marked reference to the next. Where multiple units reference the declaration, a menu of units is present and the definition operation applied to any of the menu entries brings up a marked image of the indicated unit. Show_Usage runs in time proportional to the number of first-level references, typically a second or two. It is an invaluable aid in determining the impact of an anticipated change.

3.3. Ada Command Language

Conventional systems typically provide some sort of command shell that executes *programs*, specially prepared and loaded collections of units that can be executed from the command shell. These procedures must either live in a simplified world without parameter passing or understand how to read arguments from the command line. Then, if a normal procedure wants to call one of these programs, it is necessary to understand how to invoke a shell and construct the parameters as if they were being passed in from the shell.

In the Rational Environment, any coded visible subprogram can be executed simply by calling it, provided that the closure of units required by Ada rules is also coded. This has a profound effect on the accessibility of code for execution and testing. By unifying the shell/program interface to use the normal Ada parameter mechanisms, the interface is made both simpler and more powerful.

One salient improvement is the ability to use the richness of Ada semantics. This ability to reference the declarations in Ada units isn't limited to procedures and functions; it extends to all Ada declarations: types, objects, constants, generics. The advantages that Ada has for the expression of application designs are available for the specification of the system interface or user-written utilities. This generality has far-reaching implications on the appearance, usage, and implementation of the system. References to procedures with complicated parameter profiles can be expressed using the name notation, parameter defaulting, and overloading. Interfaces to predefined packages, e.g.,

---

[4] It is also possible to type its name if there is no immediate occurrence to point to.

Text_IO, are just as easily invoked as commands to create files, using the same interface.

The full power of Ada is important to making the command interface work. An Ada-like interface that is limited to normal command-style entries might seem an attractive tradeoff between generality and implementation effort, but closer inspection reveals the limitations of this strategy. Cutting isolated features from any language is a treacherous undertaking. As a simple example, private types are useful for providing abstract interfaces to system functionality, and having private types in the command interface turns out to be quite useful. This usefulness, in turn, depends on the ability to provide function results as parameters and, in many cases, to make them default reasonably.

The command interface is an Ada declare block into which the user typically enters a single procedure call that is executed. In the general case, it is possible to write complete Ada programs using tasks, generics, or any other Ada construct in this block. The block is then compiled, and code is generated and executed.[5] The completeness of the facility is often exploited in learning Ada and determining "what would happen if ...". Since the interface is Ada, it is strongly typed; it benefits from detection of errors during compilation rather than during execution; syntactic and semantic completion are provided.

3.4. Editor-Based User Interface

All interactions with the Rational Environment are through a general, object-oriented, multi-window editor. At one level, the editor provides familiar "what you see is what you get" on the images corresponding to the objects being edited or viewed. The text of the images can be modified directly using character, word, and line operations; portions of images can be copied or moved to other locations in the same or different images; there is a general search/replace interface. All of these capabilities allow the user to view and modify objects in a human-readable form: text.

Many of the various types of objects in the system, most notably Ada, are stored in more interesting data structures than text. To support the transition from text editing to object representation, the editor supports an incremental change, parse, pretty print cycle. Changes to the text are saved for processing by type-specific editors that understand the syntax of the particular object. The changes are processed by the incremental parser to create consistent object structures. As necessary, the revised data structures are reflected onto the image with any corrections or embellishments that are deemed appropriate by the editor for the type. A type-specific editor, called an *object editor*, is available for each of the main object types. All of these implement similar editing cycles, but the operations, grammar, and semantics for Ada, discussed below, are the most interesting.

The actual operations provided for editing an object are logically separated into three classes: image operations, common object operations, and type-specific operations. Image operations are the outer-level, character-oriented operations; these are the same for all object types. Common operations are those that are expected to be available for all object types, but depend on the characteristics of the type; these include edit, structural selection, detail control, and various state transformations. Type-specific operations are provided by some types of objects where the characteristics of the type require additional functionality. Creating an object is type-specific.

---

[5] There is a special fast path provided for a common subset of known procedures for which no code is generated. This covers about 80% of the command executions. Users are typically unaware of which path a particular command takes.

A simple, but commonly used, object editor is the one provided for the subclass of files corresponding to text. Its use with files, whether created by the editor or written by programs with Text_IO, is fairly conventional but benefits from the ability to select text from other object types for inclusion in documents, mail, or bug reports. The text object editor is also responsible for dealing with Standard_Input and Standard_Output for interactively executed programs. In this mode, the user has full access to the features of the editor in providing input to programs and scanning their output, either while they are running or long after they have completed.

One of the features of the editor interface is that it doesn't impose any particular interaction sequence on its users. As a result, it is possible to freely switch between objects being edited and executing programs. The input required by an executing program can be provided by copying the text from another object or from a previous run of the same program. To support multiple concurrent activities, all visible windows are kept current with the values of the underlying objects, including (optionally) scrolling windows into which program output is being generated. This makes it convenient, for instance, to maintain a window on a long-running command to monitor its progress while continuing to get work done on something else.

3.5. Ada Editing

Ada was designed to allow the specification and construction of complex systems that could be read, understood, and maintained. A person has to write the programs, preferably using the expressive capabilities that will serve well throughout the life of the code. The purpose of the Ada object editor is to make the writing as easy as possible.

By understanding the syntax of Ada, the editor is able to provide interactive syntax checking and completion. Syntactic completion is based on the notion that many tokens in the syntax are redundant; providing the additional tokens is only marginally harder than detecting their absence. For instance, most of the structures of Ada syntax are signaled by keywords or punctuation that bracket constructs; e.g., the existence of the keyword *if* implies the future existence of *end if* and at least one statement in between. The editor uses this information to provide the keyword structure and, if required, prompts for the expression and statement portions of the statement. The result is logically very similar to operations provided by syntax-directed editors, but is stylistically similar to normal text editing and only enforces syntactic correctness at user-specified points in the editing process. Used frequently, the program can be kept syntactically correct; when necessary, wholesale editing can take place without incurring checking overhead until the changes are believed to be complete. Prompts are presented in a special font and obligingly disappear when typed over, providing convenient reminders of code still to be written. Any attempt to execute a prompt raises an exception.

A less frequently used, but powerful form of syntactic completion is provided to construct skeletal bodies for the visible operations of a unit. Completion saves typing the same procedure headers in both the visible part and the body. A related operation creates a private part with prompted completions for each of the private types in the package.

The logical extension of syntactic completion is *semantic completion*. Semantic completion fills out the contents of expressions, most commonly subprogram calls or aggregates, in a manner analogous to the way syntactic completion fills in the structural parts of the language. When making an incremental change in an installed or coded unit, it is possible to enter part of an expression, typically a procedure or function call, and request that the system fill in the parameter profile with prompts for parameters

without defaults. In doing so, the system will provide the full name-notation presentation of the call, supporting good stylistic use of the language without requiring the user to do the additional typing.

3.6. Debugging

The Rational Environment supports debugging in the same spirit as the other parts of the programming process. Debugging a program is just like running it without the debugger, except that a different "execute" key is used. No special preparations are required to set programs up to be debugged. Debugging is not intrusive: two people can be debugging the same program at the same time without getting in each other's way. Interaction with the debugger is at the source level. Program locations are displayed by bringing up the Ada image of the statement and highlighting it. Variables and parameters can be displayed by selecting them and pressing the "Put" key or by entering a command with the name of the desired variable. The value displayed is presented as it would appear in program source: record values are printed as aggregates with field names; enumeration values are printed as the appropriate enumeration literal.

3.7. Host-Target Support

Although the R1000 provides an attractive environment for the execution of Ada programs, the system was designed to support the development of programs that would run on other targets, not to be a target itself. With the exception of the execution interface, the system provides all of the facilities described for target development.

Editing and compilation appear the same for other targets as for the R1000. Indeed, the target being compiled for is a declarative property of the library and affects the content, but not the form, of the basic operations. Since we don't expect that Rational will supply code generators for every possible target, there is a general compilation interface that captures target dependencies in installation and coding, without user intervention.

Execution and debugging are less easily specified, but the debugger architecture includes support for the same set of operations on targets connected by communication lines as for native R1000 programs. There is also provision for target-specific debugging operations in a manner analogous to that used by the editor to provide type-specific operations. A variant of this host-target strategy was used successfully in debugging the Environment in its early stages.

3.8. Configuration Management and Version Control

Supporting an object-oriented view of Ada units implies support for configuration management and version control within the same integrated context. Previous experience with research environments suggested that programs need not be files, but all of these efforts focused on lone developers on prototype systems, not teams producing a product. Conventional systems solve the problem by separating program storage and version control from compilation; this separation is impractical without compromising compilation, completion, and other facilities.

A separate, but related, problem that arises in a large system is control over the configuration to be compiled and executed. Early experience showed that the connectivity of a large Ada system (the environment itself) makes it attractive to break the system up into subsystems to allow changes in one part of the system to be tested before being used by another. Simply executed, this strategy provides some relief, but it still strains compilation resources at integration points. This strain was especially bothersome, since integration took place during a prototyping stage when long delays in reintegration were undesirable.

The solution to this configuration problem was to structure subsystems to have the equivalent of visible parts and bodies. Subsystem interfaces, a subset of the visible units of the subsystem, provide the correspondent of visible parts. The complete set of units corresponds to the body of the subsystem. As with Ada units, the contract made by the visible part must be fulfilled by the body, but the implementation of the body can be changed without recompilation of clients of the visible part. An extension to the notion of incremental change of visible parts within a subsystem is that of *upward-compatible changes*. Upward-compatible changes are additional declarations that can be added to the subsystem interfaces such that references compiled against a version of the interface without the new declarations will continue to work, but new code can start to reference them.

One very effective addition to the subsystem technology was the ability to hide the private parts of packages.[6] Private parts are instrumental in providing abstract interfaces whose underlying implementation can be changed without rewriting referencing code. This extension makes it possible to change the representation without recompiling, just as if the completion of the type were in the body. For our code, this capability was particularly useful. It is common to have a package that exports private types whose completions are types exported from instantiation(s) of generics that are only referenced for this purpose. Closing the private part makes it unnecessary for the interface to appear to *with* the package exporting either the generics or the types involved. Reducing the *with* closure reduces the size of the interface while reinforcing the spirit of abstract interfaces.

This ability to compose a system of compatible subsystems that have not been directly compiled together greatly facilitates integration, especially since the assurance of semantic integrity is not lost. It does not directly address the version control problem, but leads to a version control policy based on a series of *views* -- configurations of the entire subsystem library, each spawned from the previous version of the view.

Experience with these mechanisms and experience with the compilation system have led to the construction of a more sophisticated form of view that combines the advantages of subsystems, reservation-model "source" management, and differential storage of changes to provide a facility that effectively combines the best of conventional version control with the advantages of subsystems for forming configurations. By managing views for the user, it is possible to provide support for these various forms of multiplicity in such a way that there seems to be more than one version only when differentiating configurations is part of the work at hand.

3.9. Life-Cycle Support and Extensibility

The goal of the Rational Environment is to support all of the life-cycle activities involved in software development. The initial implementation effort has focused on support for detailed design through maintenance and on building an environment that is conducive to extending these facilities into other parts of the life cycle. Our experience has been that Ada, by itself, provides a useful basis for program design, especially where it is possible to compile the designs and trace through the dependencies.

Many of the facilities that make the Environment attractive for programming also make it attractive for tool development and use. The access to DIANA and semantic information holds out the promise of building tools to analyze programs and and their development. The ability to construct interactive, editor-based interfaces has proved

[6] The R1000 architecture provides efficient support for this form of truly private type.

attractive and has helped in the process of providing useful interfaces for interesting funetionality.

4. Experience

The Rational Environment itself consists of about 800,000 lines of Ada. Development of the Environment also required building about 700,000 lines of Ada to provide cross-development tools (compilers, debuggers) and hardware/microcode development tools (simulators, translators, analysis programs). The product was first shipped to customers in February of 1985. Several significant upgrades (involving greatly improved performance, increased functionality, and improved robustness) have been delivered since then.

This development has provided considerable experience in the use of Ada with modern software engineering practices. This experience can be summarized by the following statements:

1. Adoption of Ada and the software engineering practices referenced earlier has been somewhat more difficult than anticipated. Significant investment in tools, training, and experience has been required.

2. The benefits are very real. Improvements in productivity and quality have been evident in all phases of development: design, implementation, integration, test, and maintenance.

4.1. Early Ada Experience

In 1981 and early 1982, a series of programs were constructed: development and simulation tools and prototypes of high-risk components of the Environment. These typically consisted of 50-100K lines of Ada.

Ada proved to be an excellent language for applying the concepts of information hiding, data abstraction, and hierarchical decomposition based on levels of abstraction. The basic package mechanism, separation of specification and implementation, and private types allowed rapid construction and modification of large, modular programs.

Ada cannot force good design, but it does capture and clarify the decomposition and connectivity of programs, allowing early detection and correction of architectural flaws in the design. Ada became our primary design tool, particularly for detail design. With experience, we were able to produce high-quality designs quite rapidly.

The interaction between semantic checking and modularity produced significant improvements in productivity. Using modularity and type structure to capture design information increased somewhat the time required to first execute the program, but it also greatly increased the chances that the first execution would be productive. New arrivals frequently complain that they aren't ever going to get the program to compile, only to come back later amazed that it worked the first time. When problems did arise at runtime, constraint checking allowed the errors to be detected early in execution. A common, effective debugging strategy is to run the program until an unexpected exception occurs; the problem is often evident with no additional information. Even when this is not the case, the modularity of most programs reduces uncertainty about interactions and allows much more rapid isolation of errors. It is also much easier to reason about the structure of programs and predict the consequences of a change.

Early experience also showed that all these wonderful benefits were not free. Ada semantic analysis is very expensive, increasing compilation times significantly relative to other languages. The early detection of interface and type-safety errors was handicapped by the use of batch compilation technology to report these errors. This confirmed our belief that an interactive environment for Ada with support for incremental compilation would greatly improve productivity.

4.2. Large-Scale Development and Integration

In 1983 and 1984, the development focus at Rational shifted from developing programs consisting of 10-100 packages to incrementally constructing and integrating a complete system made up of 30-40 subsystems, where each subsystem was the size of one of the earlier programs.

The system was decomposed hierarchically into five major layers, with each layer consisting of 5-8 subsystems. Although there were significant structural and interface changes over the life of the project, the basic architecture was surprisingly stable. This architecture allowed considerable parallelism in the overall development process and was instrumental in the evolution of our understanding of the configuration management and version control issues in developing large Ada systems.

At a very early point, the components of the system (or skeletons of the components) were integrated into a complete system. This initial system had very limited functionality, but allowed the basic architecture to be "debugged" before the entire system was constructed. This integration allowed system design issues such as storage management, concurrency, and error handling to be addressed very early in the development process. Early integration also served to stabilize major interfaces.

Development of the individual subsystems proceeded in parallel, with periodic integration to provide a new baseline for further development. The use of hierarchical decomposition allowed enough independence for development to proceed in parallel, while providing tight interface control to minimize integration problems. It was this integration process that led to the evolution of the subsystem concepts and supporting tools described in section 3.8.

The combination of the Ada language with object-oriented design techniques, tool support for integrating configuration management and compilation management, and an incremental integration strategy proved very effective for this particular project.

4.3. Maintenance

The Rational Environment has been in field use for about 16 months in multiple releases. Supporting it has provided some limited insight into the maintenance phase of a large Ada system. At Rational, maintenance is the responsibility of the original development team; it was crucial that new development proceed in parallel with maintenance without significant increase in development staffing.

Our experience has indicated that Ada's greatest value may be in maintenance. In this particular case, *maintenance* included bug fixes and minor enhancements, addition of major new functionality, redesign and reimplementation of several subsystems to improve performance, and reorganization of parts of the user interface. Since initial product introduction, not only has it been possible to provide desired new functionality, but reliability and robustness have improved and overall system performance has been increased by at least a factor of 3.

Efforts to improve performance are interesting examples of both the power and the

associated dangers of modularity and abstraction. Bringing up a large system in minimum time was greatly facilitated by abstract interfaces and the ability to reuse code (Ada generics). There were several cases where performance-critical sections of code were operating through generics in multiple layers of the system, where a much faster implementation was possible. Ironically, the same modularity and abstraction that introduced the problems contributed to the solution of the problems: these sections were completely redone and integrated into the system without major disruption. Abstraction is not an end in itself, but used carefully, it can help produce reliable, maintainable software to meet performance constraints.

4.4. Experience Using the Rational Environment

Our experience using the Rational Environment has confirmed those advantages we foresaw when we started the project. Interactive syntactic and semantic information makes a tremendous difference in the ease of constructing programs and making changes to them. The ability to follow semantic references makes it easier to understand existing programs and the impact of changes. The integrated debugger makes it much easier to find bugs and test fixes quickly. Taken together, these facilities have helped greatly in reducing the impact of ongoing maintenance on our ability to produce new code. We anticipate similar improvements as we achieve the same level of integration and interactivity for configuration management and version control.

The Environment has also proved useful in introducing new personnel to the project and existing personnel to new parts of the system. New personnel benefit from the assistance with syntax and semantics; everyone benefits from the ability to traverse and understand the structure of unfamiliar software. It is often possible for someone completely unfamiliar with a body of code to use these facilities to understand it well enough to successfully diagnose and fix bugs in a matter of minutes.

Acknowledgments

The design and implementation of the Rational Environment was a group effort involving too many people to mention individually. Each member of the team contributed invaluable ideas, effort, and experience. It has been a challenging, rewarding, and enjoyable process.

References

1. *Reference Manual for the Ada Programming Language.* Washington, D.C., 1983. United States Department of Defense.

2. J.E. Archer, Jr. *The Design and Implementation of a Cooperative Program Development Environment.* Ph.D. Th., Cornell University, August 1981.

3. M. Barbacci, A. Habermann, and M. Shaw. "The Software Engineering Institute: Bridging Practice and Potential". *IEEE Software 2*, 6 (November 1985), 4-21.

4. O-J. Dahl and K. Nygaard. "SIMULA - An Algol-Based Simulation Language". *Comm. ACM 9*, 9 (September 1966).

5. D.L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules". *Comm. ACM 15*, 3 (December 1972).

6. A. Evans, K. Butler, G. Goos, W. Wulf. DIANA Reference Manual. TL 83-4, Tartan Laboratories, Pittsburgh, Pa., 1983.

7. L. Druffel, S. Redwine, and W. Riddle. "The STARS Program: Overview and Rationale". *IEEE Computer 16*, 11 (November 1983), 21-29.

8. J. Ichbiah. "Rationale for the Design of the Ada Programming Language". *SIGPLAN Notices 14*, 6 (June 1979). Part B.

9. B. Liskov, and S. Zilles. "Specification Techniques for Data Abstractions". *IEEE Trans. on Software Eng. SE-1* (March 1975).

10. M. McIlroy. Mass-Produced Software Components. In *Software Engineering Concepts and Techniques*, NATO Conference on Software Engineering, 1969.

11. T. Standish and T. Taylor. Initial Thoughts on Rapid Programming Techniques. Proceedings of the Rapid Prototyping Conference, Columbia, MD, April, 1982.

12. *Requirements for Ada Programming Support Environments (Stoneman).* Washington, D.C., 1980. United States Department of Defense.

13. W. Teitelman. A Display Oriented Programmer's Assistant. CSL77-3, Xerox PARC, 1977.

14. R.T. Teitlebaum and R. Reps. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. 79-370, Cornell University, Department of Computer Science, 1979.

15. T. Wilcox and H. Larsen. The Interactive and Incremental Compilation of Ada Using DIANA. Rational, Mountain View, CA.

16. R. Yeh. "Survey of Software Practices in Industry". *IEEE Computer 17*, 6 (June 1984).